Cascades: Scalable, Flexible and Composable Middleware for Multi-modal Sensor Networking Applications

Jie Huang, Wu-chi Feng, Nirupama Bulusu, Wu-chang Feng
Portland State University, Portland, OR 97207
{jiehuang, wuchi, nbulusu, wuchang}@cs.pdx.edu

ABSTRACT

This paper describes the design and implementation of *Cascades*, a scalable, flexible and composable middleware platform for multi-modal sensor networking applications. The middleware is designed to provide a way for application writers to use pre-packaged routines as well as incorporate their own application-tailored code when necessary. As sensor systems become more diverse in both hardware and sensing modalities, such systems support will become critical. Furthermore, the systems software must not only be flexible, but also be efficient and provide high performance. Experimentation in this paper compares and contrasts several possible implementations based upon testbed measurements on embedded devices. Our experimentation shows that such a system can indeed be constructed.

Keywords: Video sensors, sensor programming, sensor management, sensor middleware.

1. INTRODUCTION

As the ability to incorporate new data types such as audio, imaging and video within the sensor network becomes possible, applications can benefit tremendously from multi-modal sets of sensors that include video. For example, environmental scientists and oceanographers are interested in the evolution of near shore phenomena along the coastal margin. Oceanographers have developed techniques to use imaging data to understand the evolution of sandbars underneath the water's surface off the coast. While they could use a massive array of in-water scalar sensors, a combination of one video sensor and some scalar environmental readings can provide the same information for their research with significant infrastructure cost savings, ease of deployment, and reduced maintenance costs[10].

While multimedia data can provide rich information for applications, a number of challenging trends are emerging. First, the diversity of data types implies diversity in the underlying hardware devices, requiring the systems software to adapt to a plethora of devices. Second, the actual in-network handling of the data becomes more application specific. Finally, the systems software must support re-tasking a deployed network. The most difficult challenge is to provide easy-to-program, application-tailored mixtures of traditional scalar sensors as well as image and video sensors while still providing a high degree of performance.

In this paper, we propose *Cascades*, a middleware system that supports composable, retaskable and application-tailorable systems software for sensor networks that contain both multimedia and scalar sensor data types. It provides a number of properties to the application. First, it provides a high-level way in which application can specify the operation of the sensor system. This includes how the data should be managed and prioritized while it is being collected. Second, it allows the user to specify application-specific algorithms (optimized in the program language of their choice) to operate on the data within the network. For example, a particular sensor application may have a specific image processing algorithm or compression mechanism in mind. Finally, it provides a way in which heterogeneous sensors can be brought together while providing reasonable performance to the application. The focus of this work will be on video support in the Cascades system. Experimental data will show that the system can be built using an application-tailored way while still providing a reasonable amount of performance.

In the following section, we briefly highlight the diversity and the systems software requirements to support such applications. In Section 3, we propose our Cascades middleware as well as several example implementations of the system. Section 4 describes a number of experiments that compare and contrast our proposed approach with other types of implementations. Section 5 provides an overview of the related work. Finally, we provide some directions for future research and conclude the paper.

2. THE VIDEO SENSING LANDSCAPE

In this section, we briefly describe some of the issues that need to be addressed for multi-modal video sensor networking applications and the ramifications on the implementation of the systems software necessary to support the system. In particular, we provide the two extremes between which we believe the software will ultimately lie.

Heterogeneity: The systems software will need to support a diversity of sensor hardware. To maximize efficiency, the systems software can export the bare minimum abstraction of the underlying hardware (e.g. TinyOS [3]) but this makes it hard to manage, program, and connect with other hardware in an application-specific way. At the other extreme, one could provide a virtual machine interface to all hardware [9]. This approach makes programming and management of the system more efficient. Given its higher overhead, however, it may not even be possible to push the abstraction to the smaller devices. Furthermore, providing virtual machine abstractions may be very inefficient for large data streams such as video because performance features such as memory mapping I/O devices may not be allowed.

Composability: Undoubtedly, the actual operation of the system will need to be tailored to a specific application's requirements. We expect that the sensor system will provide (i) a number of pre-defined components that can be used, (ii) mechanisms to support the addition of new components, and (iii) the ability to combine the components in a meaningful way. At one extreme, composability can be accomplished through pre-defined code segments that are compiled together into a single monolithic executable, allowing the system to run as efficiently as possible, while making changes to a running system more difficult. At the other end of the spectrum, one could imagine using a shell-level scripting program to compose such a system together from a number of smaller executables. While making it easier to distribute smaller sub-components, the system may suffer from a large amount of overhead in switching between address spaces and marshalling of data between executables.

Adaptability: The system will need to adapt to a number of conditions including available computation, networking availability, and power. For example, an embedded device may be able to capture video but may be constrained in its ability to compress or transmit it. To provide the most efficient operation, one could tune the sensor software to capture, compress, and transmit as much information as the smallest bottleneck in the system can handle. Furthermore, one could hard code exactly how the system should respond to a number of external events such as network congestion and variable power generation (e.g. solar panel). Clearly such a system would be hard to retask or specialize to a new application. At the other end of the spectrum, the individual components could "self-adapt" or infer the amount of data that ought to generated, stored, or thrown away through indirect measures (e.g. the network buffer is getting full). While such a system may not be optimal in its operation, it is easier to maintain and tune in place.

System performance: The optimization of data flow through a sensor can have a tremendous impact on its power usage and its performance. As an example, consider the Panoptes video sensor [2]. Using a plethora of optimizations including memory mapping the camera device into the address space, using compression across the USB 1.0 interconnect, and using the Intel Performance Primitives, the sensor is able to achieve the capture and compression of approximately 24 320x240 frames per second on a 3 Watt, 200-MHz embedded device. The Intel Performance Primitives nearly tripled the frame rate achievable for DCT-based video compression. These primitives, however, are designed for only one processor architecture. As a result, one can highly optimize the code for a particular hardware platform and camera combination, but it may not be suitable for any other hardware and camera combination. Using generic interfaces without much optimization yields only a handful of frames per second with nearly the same code.

Mobility: While not currently in the sensor networking landscape, having dynamic entities such as robots or other UAVs interacting with static sensor components to achieve a more complex goal will eventually be the future of sensor networking deployments. Of course, the dynamic aspect of such systems will place an even further burden on the systems software that must support it.

Clearly, the creation of next generation sensor systems will need to manage the conflicting goals of performance and management. We will propose middleware for such a system in Section 3.

3. CASCADES INFRASTRUCTURE

We propose Cascades, a middleware system to support the diversity of sensor networking applications, while providing reasonable system-level performance. We considered a number of options with the issues we outlined in Section 2 in mind. Sensor code compiled into one executable was ruled out because updating the functionality of a video sensor would require a significant amount of wireless bandwidth to be used to distribute all the code, whether it changed or not. One could also use shell-level scripting, connecting individually compiled pieces of code to be brought together. This

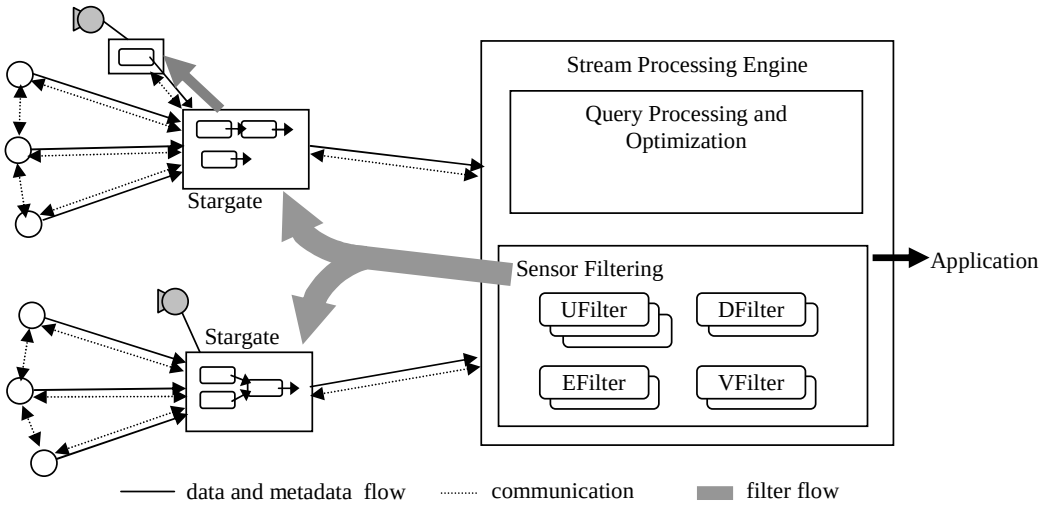

FIGURE 1: This figure shows an example of the overall architecture for our proposed sensor networking middleware. The nodes labeled *Stargate* are slightly more powerful, in-network, sensor nodes that can both capture video and be used to manage a number of scalar sensor nodes.

solution was also ruled out as not having high enough performance. The other alternative was using a high-level scripting language like TCL or Python. The key advantage of such languages is that they are interpreted scripting languages, allowing users to specify rather complex systems with minimal code. Furthermore, they allow programs written in high-level languages such as C or C++ to be called as part of the script. This allows a majority of computationally intensive code (such as video processing algorithms) to be written in a highly optimized way.

3.1 The CASCADES Infrastructure

Our middleware uses Python-based interfaces to connect filters together. Filters can be constructed from and attached to other filters to specify actions that may be taken on the data flowing through the filter. The basic concepts of building a system out of TCL and Python are similar. We chose Python over TCL for several reasons. First, it provides more complex data structures. Second, it is more efficient, which is extremely important for power-constrained sensor systems. Third, Python provides the ability to add or change the behavior of parts of the system while it is running. As a result, for the parts of the system that are connected via Python, re-tasking the system involves distributed the new code segment and updating the script so that it points to the new code (e.g. a new video compression algorithm)[a]. Fourth, Python interfaces also provide the opportunity to provide type checking of the data so that the components that are plugged together can be verified for compatibility, if so desired. Finally, it is easier to construct more complicated programs in Python, giving the user more control over the system rather than hiding many details. The last point is both a positive and negative. In the hands of more experienced programmers, Python is easier to adapt to application specifics.

3.2 Constructing Cascades

The primary mechanism to support the management and integration of multi-modal data are cascading filters. The filters can be constructed at a number of levels. Filters are user-supplied or toolkit-derived functions that allow the sensor system to tailor its data for the user application. The idea of each filter is that it allows the processing of the data within the filter to be accomplished with a highly optimized piece of code (rather than an interpreted language). There are several basic types of filters that we envision:

Efilters are the primary mechanism by which the handling of faulty sensors can be specified. Faulty readings can occur from bio-fouling of the sensors in outdoor scenarios. These filters can consist of standard statistical filtering techniques in a default-model as well as allow the application user to specify the exact way in which the faulty data may be handled.

Dfilters are used to manage scalar data within the sensor network. They take one or more streams of scalar dat and produce an output of one or more data streams as well as meta-information about the sensor data. As an example, one filter might calculate the average value measured per hour, either for a single sensor or a group of sensors. The filter

[a] This assumes that the Python script periodically checks whether or not the script has been updated.

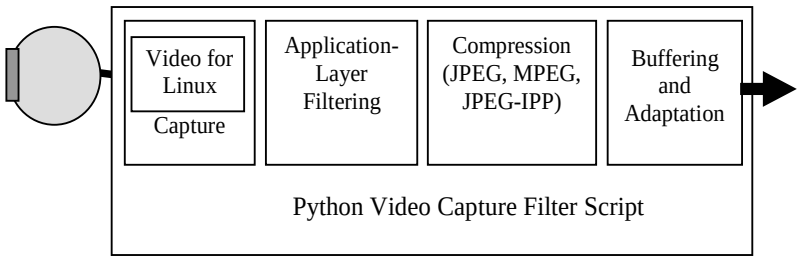

FIGURE 2: Video capture filter. This figure shows the construction of the video sensor capture and compression system that we have built. Each of the subcomponents has a Python interface, allowing it to be arranged in a variety of ways. The Python script for this example video capture system is shown in Figure 3.

might also add meta-information such as timing information or relational information between sensors. The sensor output can then be used by other filters. As we will describe later,

Vfilters are used to manage video data being collected by video sensors. Vfilters might consist of application specific video processing algorithms or off-the-shelf components. Application-specific algorithms may include image processing techniques for the environmental monitoring example previously described. An off-the-shelf component might include a compression algorithm or video adaptation algorithm within the network.

Ufilters are user specified filters that allow the user to specify the integration of data from the other types of filters. These can include annotation of video streams using scalar sensor data.

Each filter supports a basic "container" mechanism with headers that describe the type of data (e.g. scalar sensor data, a compressed video frame, a raw image, etc.) along with the size of the data. The Python subsystem can then be viewed as a message passing interface that shuttles data in between the various filters. Filters are obviously a somewhat loose term that implies some basic functionality. The filters can be composed of smaller filters that wrap some optimized functionality and can, therefore, provide some hierarchical organization of the filtering mechanism.

The overall Cascades infrastructure is shown in Figure 1. The actual hardware devices are abstracted just enough to provide data into one of the filters. For example, scalar data sensors will have the operating system of their choice running on the system and will talk upstream to a node (such as a Crossbow Stargate device) that will provide the Python-based abstraction. We have built a filter that encapsulates the functionality of TinyDB on the Stargate and that exports the data through our generic python interconnect.

3.2.1. Video Filters

As previously mentioned, filters can be constructed out of other "filters". In the remainder of this section, we will describe how we have constructed the systems code for a video sensor. As described in the original *Panoptes* paper [2], the video sensor code consists of a number of smaller modules including: video capture, application-specific filtering, compression, and buffering and adaptation. We have re-written each of these modules so that they are exportable into Python interconnects. The overall video sensor architecture is shown in Figure 2. Each of the modules adheres to a message format containing items such as the type of data within the message, the length of the message, and the data. The actual interface to Python can be hand generated or done via an automatic interface generator such as SWIG. We will describe these details in the following section.

The actual Python script for a video sensor is shown in Figure 3. The top section of code is for initialization purposes and imports the C modules that will be used in the video capture and compression filter. The second section of code initializes the camera, buffer, and modification time of the Python script. The modification time is used by the Python script to automatically reload the components if the Python script gets modified. This is an important aspect of our system as it allows (i) a new piece of sensor code (such as a new compression algorithm) to be loaded into the sensor and (ii) allows the system to modify its behavior without having human intervention. The third part of the code is used to initialize the interconnections and the network connection to the upstream camera manager and sink (as described in the Panoptes paper [2]). Finally, the last part of the code is the actual script for running the video sensor. As shown by the code in Figure 3, there are relatively few lines of code necessary to construct the system and these components can be connected in other ways as the objects are relatively generic in calls.

VideoCapture.py

```
import os, sys
from stat import *
from cam import *
from secretary import *
from messenger import *
import filters

modify_time = os.stat("filters.py")[ST_MTIME]
camera = Camera()
buffer = MsgBuffer(10, 100)
camera.Initialize()

messenger = Messenger("capture")
secretary = Secretary()
messenger.SetPrivMessenger(secretary)
messenger.AddSocket("sink", 2183, "10.0.0.1")

def run():
  global modify_time
  messenger.PollSockets()
  raw_image = camera.CaptureOneFrame()
  JPEG_msg = filters.run(raw_image)
  buffer.PutMsg(JPEG_Msg)
  msg2send = buffer.GetNextMsgToSend()
  if messenger.SendMsg("sink", msg2send,-1):
    buffer.RemoveSelectedMsg()

  new_time = os.stat("filters.py")[ST_MTIME]
  if modify_time != new_time:
    modify_time = new_time
    reload(filters)
```

FIGURE 3: Video Capture Python Script. This figure shows the Python script necessary to combine the various components of the video sensor (as shown in Figure 2). The `modify_time` lines are used by the Python script to watch for the modification of the filter, allowing the Python script to reload upon being modified (e.g. having a new compression algorithm installed).

VideoManager.py

```
#Initialization and minor procedures removed.

def is_time_to_resume_receiving():
  global buffer_full, buffer
  if buffer_full and buffer.HasSpareSpace(8):
    buffer_full = 0
    return 1
  return 0

while not end_of_experiment():
  messenger.PollSockets()
  update_dropping_level()
  msg = secretary.GetMsgFrom("sensor1")
  buffer.PutMsg(msg)
  msg = secretary.GetMsgFrom("sensor2")
  buffer.PutMsg(msg)

  if is_time_to_stop_receiving():
    cmd = secretary.MakeStopSendingMsg()
    messenger.SendMsg("sensor1", cmd, -1)
    messenger.SendMsg("sensor2", cmd, -1)

  if is_time_to_resume_receiving():
    cmd = secretary.MakeStartSendingMsg()
    messenger.SendMsg("sensor1", cmd, -1)
    messenger.SendMsg("sensor2", cmd, -1)

  if network_on():
    msg2send = buffer.GetNextMsgToSend()
    if messenger.SendMsg("sink",msg2send,-1):
      buffer.RemoveSelectedMsg()
```

FIGURE 4: In-network video manager. This code show the construction of an in-network buffer management routine. It manages two video sensors, which are sending prioritized frame information to it. The functions is_time_to_stop_receiving() and network_on() are 4-8 line Python code segments similar to is_time_to_resume_receiving() (removed for space reasons).

3.3 A Larger Example System

The power of the Python-based infrastructure is that it allows more complex systems to be built out of smaller components. As an example, we have built a hierarchical buffering algorithm for a video data aggregator within the sensor network. We focus on a three-node system. Two of the nodes are the Panoptes video sensors [2] running the code shown in Figure 3 and the third is a video manager in the middle of the network. The idea is that the video manager can provide a common buffer shared amongst the other sensors. When the common buffer is full, two video sensor nodes should stop sending data to the manager node to save energy. The basic code for the manager node is shown in Figure 4. Python functions is_time_to_stop_receiving() and is_time_to_resume_receiving() specify buffer management polices in a few lines and can be changed easily. The key idea to take away is that the in-network video manager can be handled through fairly high-level actions but has sufficient access to data to allow the programmer to apply application-specific algorithms in the code.

4. EXPERIMENTATION

4.1 Experimental Setup

To understand the performance of the proposed Cascades middleware, we measured the amount of overhead introduced by connecting the system via generic interfaces and the amount of extra space on the sensor needed to hold the code and Python executables, given the memory premium on such devices.

	JPEG				JPEG-IPP				MPEG			
	C	Python Native	Python-SWIG	Shell	C	Python Native	Python-SWIG	Shell	C	Python Native	Python-SWIG	Shell
160x120	29.60	29.55	29.57	27.09	29.69	29.41	29.88	28.68	22.55	21.96	21.43	20.25
320x240	10.01	10.00	9.45	8.07	18.37	18.38	17.74	13.95	8.46	8.32	8.35	7.55
640x480	2.62	2.59	2.60	2.07	5.04	5.04	5.04	3.77	2.41	2.45	2.40	2.18

FIGURE 5: This table shows the performance of the libJPEG code, JPEG code optimized by IPP, and ffMPEG MPEG-1 code using the four interconnect techniques. The numbers shown are in frames per second

We used the Crossbow Stargate embedded sensor platform. The Stargate is meant to be a higher-level processing board for scalar sensors. As such, it is a natural place in the sensor hierarchy in which to start using higher-powered video sensing. The Stargate platform we have runs the embedded Linux operating system 2.4.19-rmk7-pxa2. The platform has a 400 MHz Intel Xscale processor, 64 Mbytes of memory, a 100 Mbit Ethernet connector, and a compact flash wireless 802.11 card. The video capture is accomplished through a Logitech QuickCam 4000 Pro USB camera [2].

For experimentation, we will compare and contrast four different types of system architectures. We have built the video sensor with a single monolithic C program. We will refer to this approach as the *C* approach. We have also built each of the components as standalone executables, using a shell script with pipes to interconnect the components. We will refer to this approach as the *Shell* approach. For the Python-based system we have two approaches. Both approaches use the same compiled C modules. One approach, referred to as the *Python-SWIG* approach, uses the Simplified Wrapper and Interface Generator (SWIG) system to generate the Python interfaces for the C code. The other approach, which we refer to as *Python-Native*, uses hand coded C to Python interface mappings. SWIG can generate necessary glue code automatically but may lead to excess code given its generic nature.

For compression performance, we have implemented three different compression algorithms in order to get an understanding of the efficacy of various compression routines on the Stargate platform. We expect such numbers to be useful in understanding what can and cannot be done in future multi-modal sensor networks and what the minimum requirements are. The three compression algorithms that we have implemented are *JPEG*, *JPEG-IPP*, and *MPEG*. The *JPEG* algorithm is based upon the standard *libJPEG* source code that is freely available. The code is optimized in a CPU independent way, and thus, represents a generic image compression algorithm. The *JPEG-IPP* algorithm takes advantage of the Intel Performance Primitives (IPP) libraries that are available from Intel. The IPP libraries provide routines for copying large amount of memory, performing DCT functions, encoding Huffman symbols, and other multimedia related tasks. The libraries are primarily low-level assembly routines that take advantage of the architecture in any way possible. The *MPEG* algorithm is the MPEG-1 video coder from *ffMPEG*, which has been relatively optimized for the StrongArm and Xscale processors when the right compile flags are chosen.

4.2 System Performance

In this section, we compare and contrast the four different approaches that we have implemented using the three different compression algorithms. The numbers were generated by capturing 300 frames with the experimental set up and then measuring the number of frames per second it was able to capture. The numbers are shown in Figure 5.

For JPEG, the system is able to keep up with the camera's capture rate at the resolution 160x120 in all cases except the Shell programming case. The multiple threads and I/O necessary to move information between shell scripted entities are non trivial, as expected. Moving to 320x240 pixel frames, we see that the C, Python SWIG and Python Native algorithms perform similarly. This is relatively encouraging as it suggests that the overhead of using SWIG is not that high. It is also noted that using shell scripting in this case requires approximately 20% overhead. Finally, in the 640x480 case, we see that the processor is completely overwhelmed with data per frame. As a result the shell scripting version performs similarly to the Python and C versions because each frame requires much more relative processing, mitigating the context switching cost for the Shell program.

For JPEG-IPP at 160x120, the Stargate processor is able to keep up with the camera's capture rate. The Shell version is slightly faster primarily due to the IPP code freeing up some of the compute cycles to do context switching. For the 320x240 pixel video, we see that the IPP-based code is able to achieve a video capture rate of nearly 80 to 87% better than its non-IPP-based counterpart. As reported previously, this suggests that in building such sensor systems, the need to support hand tuning on specific platforms is critical to performance. As a result, systems that attempt to abstract

	C	Python Native	Python-SWIG	Shell
Compiled Code	95.7	184.8	299.6	145.0
Script	--	1.0	0.6	0.6
Interpreter	--	1103.7	1103.7	--

FIGURE 6: Code Sizes. This table shows the size in kilobytes of the various subcomponents for the JPEG-IPP

	C	Python Native	Python-SWIG	Shell
Compiled Code	1064.9	1072.0	1318.3	1114.1
Script	--	1.2	0.6	0.6
Interpreter	--	1103.7	1103.7	--

FIGURE 7: Code Sizes. This table shows the size in kilobytes of the various subcomponents for the MPEG-based

video devices should probably be avoided. For the 320x240 pixel case, we again see that the C and various Python versions are similar again. The Shell version increased its overhead from 14% to 22%. As the CPU is fairly saturated at this resolution, the context switching overhead is higher relative to the JPEG version. Finally, we see that in the 640x480 case, the IPP version allows for nearly a doubling of the frame rate achievable.

For ffMPEG MPEG-1 video compression algorithm, it is interesting to note that adding motion compensation between frames requires approximately 50% overhead in the 320x240 pixel and 640x480 pixel case compared to JPEG-IPP. We believe that this is partially due to the slower memory hierarchy of the embedded processor. Another point worth mentioning is that the Shell version does relatively better in the MPEG than the JPEG cases. This is entirely due to the fact that there is a significantly higher computation per frame requirement than in the JPEG cases, allowing the overhead to be amortized over more cycles.

In general, we found that the Python-SWIG and Python-Native algorithms had very similar performance. This is not entirely unexpected as the amount of marshalling of data is fairly minimal (as designed). We would expect that if the users want access to the data in Python that a higher overhead would manifest itself. We also found that the Python versions perform similarly to the C versions of the code. Given its interpretive nature, we believe that this is a significant achievement for the writers of Python and something that we should take advantage of for composability and retasking of sensor networking code. Finally, we note that it appears that the trade-off between using JPEG and MPEG is approximately 50% for the 320x240 case. It would appear that for extremely bandwidth-stringent environments such a trade-off may be worth making.

4.3 Code Size

The one potential drawback of using Python is that it requires that the Python interpreter be installed on the each machine running the Python scripts. Clearly, this could limit the types of embedded processors that the code can run on. In Figure 6, we have listed the code sizes for the JPEG-IPP algorithm. Here, we see the clear differences between the various approaches. The C code is a single compiled object allowing all the standard libraries to be compiled in just once. As a result, the Shell connected code is approximately 51% larger than the C code. The Python versions require even more space. This is mostly due to the interface specification between C and Python. Writing the interface between C and Python (i.e. Python-Native) saves 120 kilobytes, at the expense of additional programming. We also see that there is approximately a 1 Mbyte overhead to store the Python interpreter as well.

In Figure 7, we have listed the MPEG-based code sizes. As shown in the table, MPEG requires significantly more space (and processing power) in order to operate on the embedded devices. Because of the relatively large size of the MPEG compiled code, much of the overhead of Python is amortized in the larger code. As the Python interfaces are fairly similar, by design, the overhead between the various approaches are constant as well.

5. RELATED WORK

Several sensor platforms have been developed with varying capabilities for use in sensor networking applications, including the Berkeley mote family of WeC, Rene, Dot, Mica, MicaZ, and XSM sensors [7]. The main characteristics of these sensors are that they have small amounts of memory, have very limited processing capabilities, and are very efficient with battery power. The Cyclops image sensor is a low-power image sensor that can be attached to a Crossbow Mica2 sensor [8]. The Panoptes video sensor (based on the Crossbow Stargate device) [2] is the only generic low-power video sensor that we are currently aware of. These developments just highlight the fact that the systems software will have to evolve to support a diversity of hardware.

Techniques such as Directed Diffusion [5] or TinyDB [4] to process scalar data in a sensor work have been developed. They provide mechanisms for simple in-network data aggregation. Cascades provides support for more complex tasks. In

addition, Cascades enables a whole-network reconfigurable deployment of filters instead of filter deployment on individual sensors. Cascades can also be used with Diffusion or TinyDB. For example, one can control a herd of motes via TinyDB, and use TinyDB's data stream output as an input to a Cascade filter.

There are many stream query processing systems for sensor networks including Cougar, Telegraph, Stream, and Aurora/Borealis [1]. One key difference between the existing stream processing work and Cascades is the inclusion of video processing and filtering within the sensor substrate as well as application-specific data handling and adaptation in the event of insufficient resources exist. We believe that much of the functionality of such stream processing engines can be incorporated into the filtering mechanisms we propose.

Finally, we note that finding efficient "plug and play" architectures for multimedia has been the focus of some previous research. The *Continuous Media Toolkit* (CMT) from Berkeley focuses on the rapid development and deployment of distributed streaming applications [6]. The CMT toolkit is a TCL/TK-based system that allows users to construct streaming applications through scripts that combine lower-level components together. The Cascades approach is similar in vein to the CMT toolkit. The sensor networks we envision, however, will force the scripting languages to manage more complex data sharing between components. We believe that Python is more usable in this context because it provides real data structures to the scripts and runs faster than TCL. The data structures are necessary to support the movement of data between the filters.

6. CONCLUSION AND FUTURE WORK

In this paper, we have outlined *Cascades*, a potential middleware system to support diversity and programmability in multi-modal sensor networks. As sensor networks continue to evolve, the systems software that supports such applications must also evolve while continuing to provide highly optimized and efficient operation. Python-based cascading filters can effectively manage the trade-off of highly optimized systems and the need to tailor the system to the user application. We have presented a number of experimental results that have shown that using Python interconnects can allow basic scripts to orchestrate the sensor network, while requiring very little overhead.

There are still a number of open issues that we are working to address with Cascades. Currently, there is no explicit mechanism to adapt computation to conserve power. This requires coordination among filters. Another avenue of future work is managing power on a larger time-frame. For example, in the oceanographic example, it may be more beneficial to store data during the day and only transmit at night (while no video capture is happening).

ACKNOWLEDGEMENTS

The authors would like to thank Edward Epp of Intel for making the Stargate platform usable for Video for Linux and the Logitech cameras, which our video sensors use. This material is based upon work supported by Intel and the National Science Foundation under Grant Nos. EIA-0130344 and CISE-RR-0514818.

REFERENCES

1. D. Abadi, et. al, "The Design of the Borealis Stream Processing Engine", in *Proc. of the Second Biennial Conference on Innovative Database Systems (CIDR'05*), Asilomar, CA, USA, January 2005, pp. 277-289.
2. W. Feng, B. Code, E. Kaiser, M. Shea, W. Feng, L. Bavoil, "Panoptes: A Scalable Architecture for Video Sensor Networking Applications", in *Proc. of ACM Multimedia*, Berkeley, CA, USA, Nov. 2003, pp. 562-71.
3. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. "System Architecture Directions for Networkded Sensors". ASPLOS. Cambridge, MA, USA, 2000.
4. C. Intanagonwiwat, R. Govindan and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks", in *Proc. of ACM MobiCOM*, Boston, MA, USA, August 2000. pp.56-67.
5. S. Madden, M. Franklin, J. Hellerstein, W. Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks", in *Proc. of OSDI*, Boston, MA, USA, Dec. 2002.
6. K. Mayer-Patel, L. Rowe, "Design and Performance of the Berkeley Continuous Media Toolkit", in *Proc. of Multimedia Computing and Networking 1997*, San Jose, CA, USA, pp 194-206, 1997.
7. J. Polastre, Design and Implementation of Wireless Sensor Networks for Habitat Monitoring, M.S. Thesis, EECS, Univ. of Calif., Berkeley, 2003.
8. M. Rahimi, D. Estrin, R. Baer, H. Uyeno, J. Warrior, "Cyclops: image sensing and interpretation in wireless networks", in *Proc. of ACM SenSYS 2004*, Baltimore, MD, USA, pp. 311, November 2004.
9. Senses Project: http://senses.cs.udavis.edu
10. H.F. Stockdon, R.A. Holman, "Estimation of Wave Phase Speed and Nearshore Bathymetry from Video Imagery", *Journal of Geophysical Research*, Vol. 105, No. C9, pp 22,015-22,033, Sept. 2000.